Kinds Are Calling Conventions

Paul Downen, Zena M. Ariola, Simon Peyton Jones, Richard A. Eisenberg
Efficient Function Calls

Parameter Passing Techniques
Efficient Function Calls

- Representation — What & Where?
Efficient Function Calls

- Representation — What & Where?
- Arity — How many?
Efficient Function Calls

• Representation — What & Where?
• Arity — How many?
• Levity (aka Evaluation Strategy) — When to compute?
Efficient Function Calls

- Representation — What & Where?
- **Arity** — How many?
- Levity (aka Evaluation Strategy) — When to compute?
Determining Function Arity

f1, f2, f3, f4 :: Int -> Int -> Int

Type suggests arity 2
Determining Function Arity

f1, f2, f3, f4 :: Int -> Int -> Int  

Type suggests arity 2

f1 = \x -> \y ->
    let z = expensive x
    in y + z
Determining Function Arity

\[ f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad \text{Type suggests arity 2} \]

\[ f_1 = \lambda x \rightarrow \lambda y \rightarrow \]
\[ \text{let } z = \text{expensive } x \quad \text{Arity 2} \]
\[ \text{in } y + z \]
### Determining Function Arity

\[
f_1, f_2, f_3, f_4 :: \text{Int} \to \text{Int} \to \text{Int} \quad \text{Type suggests arity 2}
\]

\[
f_1 = \lambda x \to \lambda y \to \\
\phantom{f_1} \quad \text{let } z = \text{expensive } x \quad \text{Arity 2} \quad f_2 = \lambda x \to f_1 x \\
\phantom{f_1} \quad \text{in } y + z
\]
Determining Function Arity

\[ f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \quad \text{Type suggests arity 2} \]

\[ f_1 = \lambda x \rightarrow \lambda y \rightarrow \]
\[ \quad \text{let } z = \text{expensive } x \quad \text{Arity 2} \quad f_2 = \lambda x \rightarrow f_1 x \]
\[ \quad \text{in } y + z \quad = \lambda x \rightarrow \lambda y \rightarrow f_1 x y \]
Determining Function Arity

\[ f_1, f_2, f_3, f_4 :: \text{Int} \to \text{Int} \to \text{Int} \quad \text{Type suggests arity 2} \]

\[ f_1 = \lambda x \to \lambda y \to \]
\[ \quad \text{let } z = \text{expensive } x \quad \textbf{Arity 2} \quad f_2 = \lambda x \to f_1 x \quad \textbf{Arity 2} \]
\[ \quad \text{in } y + z \]
\[ = \lambda x \to \lambda y \to f_1 x y \]
Determining Function Arity

\( f_1, f_2, f_3, f_4 :: \text{Int} \to \text{Int} \to \text{Int} \) \hspace{1cm} \text{Type suggests arity 2}

\( f_1 = \lambda x \to \lambda y \to \)
\[
\begin{align*}
\text{let } z &= \text{expensive } x \quad \text{Arity 2} \\
\text{in } y + z
\end{align*}
\]

\( f_2 = \lambda x \to f_1 x \quad \text{Arity 2} \)
\[
\begin{align*}
\text{in } \lambda y \to y + z
\end{align*}
\]

\( f_3 = \lambda x \to \)
\[
\begin{align*}
\text{let } z &= \text{expensive } x \\
\text{in } \lambda y \to y + z
\end{align*}
\]
Determining Function Arity

\( f_1, f_2, f_3, f_4 :: \text{Int} \to \text{Int} \to \text{Int} \)  \hspace{1cm} \text{Type suggests arity 2}

\[
f_1 = \lambda x \rightarrow \lambda y \rightarrow \\
\quad \text{let } z = \text{expensive } x \hspace{1cm} \text{Arity 2} \\
\quad \text{in } y + z
\]

\[
f_2 = \lambda x \rightarrow f_1 x \hspace{1cm} \text{Arity 2} \\
\quad = \lambda x \rightarrow \lambda y \rightarrow f_1 x \hspace{1cm} y
\]

\[
f_3 = \lambda x \rightarrow \\
\quad \text{let } z = \text{expensive } x \\
\quad \text{in } \lambda y \rightarrow y + z
\]

Hint: ‘expensive \( x \)’ may be costly, or even cause side effects
Determining Function Arity

\[ f_1, f_2, f_3, f_4 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]  

Type suggests arity 2

\[ f_1 = \lambda x \rightarrow \lambda y \rightarrow \]
\[ \quad \text{let } z = \text{expensive } x \quad \text{Arity 2} \]
\[ \quad \text{in } y + z \]
\[ \]
\[ f_2 = \lambda x \rightarrow f_1 x \quad \text{Arity 2} \]
\[ = \lambda x \rightarrow \lambda y \rightarrow f_1 x \quad y \]

\[ f_3 = \lambda x \rightarrow \]
\[ \quad \text{let } z = \text{expensive } x \quad \text{Arity 1} \]
\[ \quad \text{in } \lambda y \rightarrow y + z \]

\[ f_4 = \]  

Hint: ‘\text{expensive } x’ may be costly, or even cause side effects
Determining Function Arity

\( f_1, f_2, f_3, f_4 :: \text{Int} \to \text{Int} \to \text{Int} \)  

Type suggests arity 2

\( f_1 = \lambda x \to \lambda y \to \)

\[
\begin{align*}
\text{let } z = \text{expensive } x & \quad \text{Arity 2} \\
\text{in } y + z
\end{align*}
\]

\( f_2 = \lambda x \to f_1 x \)  

\( \quad \text{Arity 2} \)

\( f_3 = \lambda x \to \\
\text{let } z = \text{expensive } x \quad \text{Arity 1} \\
\text{in } \lambda y \to y + z \)

\( f_4 = \lambda x \to f_3 x \)

Hint: ‘expensive x’ may be costly, or even cause side effects
Determining Function Arity

\( f_1, f_2, f_3, f_4 :: \text{Int} \to \text{Int} \to \text{Int} \)  \quad \text{Type suggests arity 2}

\[ f_1 = \lambda x \to \lambda y \to \text{let } z = \text{expensive } x \quad \text{Arity 2} \quad f_2 = \lambda x \to f_1 x \quad \text{Arity 2} \in y + z \]
\[ f_3 = \lambda x \to \text{let } z = \text{expensive } x \quad \text{Arity 1} \quad f_4 = \lambda x \to f_3 x \]
\[ \text{in } \lambda y \to y + z \quad \neq \lambda x \to \lambda y \to f_3 x y \]

Hint: ‘expensive \( x\)’ may be costly, or even cause side effects
Determining Function Arity

\[ f_1, f_2, f_3, f_4 :: \text{Int} \to \text{Int} \to \text{Int} \quad \text{Type suggests arity 2} \]

\[ f_1 = \lambda x \to \lambda y \to \]
\[ \quad \text{let } z = \text{expensive } x \quad \text{Arity 2} \]
\[ \quad \text{in } y + z \]

\[ f_2 = \lambda x \to f_1 x \quad \text{Arity 2} \]
\[ = \lambda x \to \lambda y \to f_1 x y \]

\[ f_3 = \lambda x \to \]
\[ \quad \text{let } z = \text{expensive } x \quad \text{Arity 1} \]
\[ \quad \text{in } \lambda y \to y + z \]

\[ f_4 = \lambda x \to f_3 x \quad \text{Arity 1} \]
\[ \neq \lambda x \to \lambda y \to f_3 x y \]

Hint: ‘expensive x’ may be costly, or even cause side effects.
What Is Arity?

For Curried Functions
What Is Arity?

Definition 1. The number of arguments a function needs to do “serious work.”
What Is Arity?

**Definition 1.** The number of arguments a function needs to do “serious work.”

- If ‘f 1 2 3’ does work, but ‘f 1 2’ does not, then ‘f’ has arity 3
What Is Arity?

Definition 1. The number of arguments a function needs to do “serious work.”

• If ‘f 1 2 3’ does work, but ‘f 1 2’ does not, then ‘f’ has arity 3

Definition 2. The number of times a function may be $\eta$-expanded without changing its behavior or cost.
What Is Arity?

**Definition 1.** The number of arguments a function needs to do “serious work.”

- If ‘f 1 2 3’ does work, but ‘f 1 2’ does not, then ‘f’ has arity 3

**Definition 2.** The number of times a function may be η-expanded without changing its behavior or cost.

- If ‘f’ is equivalent to ‘\( x \ y \ z \rightarrow f \ x \ y \ z \)’, then ‘f’ has arity 3

For Curried Functions
What Is Arity?

For Curried Functions

**Definition 1.** The number of arguments a function needs to do “serious work.”

- If ‘f 1 2 3’ does work, but ‘f 1 2’ does not, then ‘f’ has arity 3

**Definition 2.** The number of times a function may be η-expanded without changing its behavior or cost.

- If ‘f’ is equivalent to ‘\x y z -> f x y z’, then ‘f’ has arity 3

**Definition 3.** The number of arguments passed simultaneously to a function during one call.
What Is Arity?

**Definition 1.** The number of arguments a function needs to do “serious work.”

- If ‘f 1 2 3’ does work, but ‘f 1 2’ does not, then ‘f’ has arity 3

**Definition 2.** The number of times a function may be η-expanded without changing its behavior or cost.

- If ‘f’ is equivalent to ‘\x y z -> f x y z’, then ‘f’ has arity 3

**Definition 3.** The number of arguments passed simultaneously to a function during one call.

- If ‘f’ has arity 3, then ‘f 1 2 3’ can be implemented as a single call
What Is Arity?

**Definition 1.** The number of arguments a function needs to do “serious work.”
- If ‘f 1 2 3’ does work, but ‘f 1 2’ does not, then ‘f’ has arity 3

**Definition 2.** The number of times a function may be η-expanded without changing its behavior or cost.
- If ‘f’ is equivalent to ‘\(\lambda x \ y \ z \to f \ x \ y \ z\)’, then ‘f’ has arity 3

**Definition 3.** The number of arguments passed simultaneously to a function during one call.
- If ‘f’ has arity 3, then ‘f 1 2 3’ can be implemented as a single call
Goal: An IL with unrestricted $\eta$ for functions, along with restricted $\beta$ for other types
Static Arity

In an Intermediate Language
Static Arity

In an Intermediate Language

- New $a \rightsquigarrow b$ type of primitive functions (ASCII ‘$a \rightarrow b$’)
  - To distinguish from the source-level $a \rightarrow b$ with different semantics
Static Arity

In an Intermediate Language

- New $a \rightsquigarrow b$ type of primitive functions (ASCII ‘$a \rightsquigarrow b$’)
  - To distinguish from the source-level $a \rightarrow b$ with different semantics
- Primitive functions are fully extensional, unlike source functions
  - $\lambda x. f x =_\eta f : a \rightsquigarrow b$ unconditionally
Static Arity

- New $a \rightsquigarrow b$ type of primitive functions (ASCII ‘$a \rightsquigarrow b$’)
  - To distinguish from the source-level $a \rightarrow b$ with different semantics
- Primitive functions are **fully extensional**, unlike source functions
  - $\lambda x. f x =_\eta f : a \rightsquigarrow b$ unconditionally
- Application may still be **restricted** for efficiency, like source functions
  - $(\lambda x. x + x)$ (expensive $10^6$) does not recompute expensive $10^6$
Static Arity

- **New** \( a \leadsto b \) **type of primitive functions** (ASCII ‘\( a \leadsto b \)’)
  - To distinguish from the source-level \( a \rightarrow b \) with different semantics
- **Primitive functions are** *fully extensional*, **unlike** source functions
  - \( \lambda x . f x = \eta f : a \leadsto b \) *unconditionally*
- **Application may still be** *restricted* for efficiency, like source functions
  - \((\lambda x . x + x) \text{ (expensive 10}^6\) does not recompute expensive 10\(^6\)
- **With full \( \eta \), types express arity — just count the arrows**
  - \( f : \text{Int} \leadsto \text{Bool} \leadsto \text{String} \) has arity 2, no matter \( f \)’s definition
Currying

When Partial Application Matters
Currying

\[ f_3 :: \text{Int} \to \text{Int} \to \text{Int} \]
\[ f_3 = \lambda x \to \text{let } z = \text{expensive } x \text{ in } \lambda y \to y + z \]
Currying

When Partial Application Matters

f3 :: Int ~> Int ~> Int
f3 = \x -> let z = expensive x in \y -> y + z

• Because of $\eta$, f3 now has arity 2, not 1!
Currying

\[ f_3 \colon \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ f_3 = x \rightarrow \text{let } z = \text{expensive } x \text{ in } y \rightarrow y + z \]

• Because of η, \( f_3 \) now has arity 2, not 1!

  • \( \text{map} \ (f_3 \ 100) \ [1..10^6] \) recomputes ‘\text{expensive } 100’ a million times 😞
Currying

\[
\begin{align*}
f_3 &:: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
f_3 &= \lambda x \rightarrow \text{let } z = \text{expensive } x \text{ in } \lambda y \rightarrow y + z \\
\end{align*}
\]

- Because of \(\eta\), \(f_3\) now has arity 2, not 1!

  - \(\text{map } (f_3 100) [1..10^6]\) recomputes ‘expensive 100’ a million times 😞

\[
\begin{align*}
f_3' &:: \text{Int} \rightarrow \{\text{Int} \rightarrow \text{Int}\} \\
f_3' &= \lambda x \rightarrow \text{let } z = \text{expensive } x \text{ in } \text{Clos}(\lambda y \rightarrow y + z) \\
\end{align*}
\]

Clos :: (\text{Int} \rightarrow \text{Int}) \rightarrow \{\text{Int} \rightarrow \text{Int}\}
Currying

$$f_3 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$
$$f_3 = \lambda x \rightarrow \text{let } z = \text{expensive } x \text{ in } \lambda y \rightarrow y + z$$

• Because of \( \eta \), \( f_3 \) now has arity 2, not 1!
  
  • \( \text{map} (f_3 \ 100) [1..10^6] \) recomputes ‘expensive 100’ a million times 😞

$$f_3' :: \text{Int} \rightarrow \{ \text{Int} \rightarrow \text{Int} \}$$
$$f_3' = \lambda x \rightarrow \text{let } z = \text{expensive } x \text{ in } \text{Clos} (\lambda y \rightarrow y + z)$$

• \( f_3' \) is an arity 1 function; returns a closure \{\text{Int} \rightarrow \text{Int}\} of an arity 1 function

\text{Clos} :: (\text{Int} \rightarrow \text{Int}) \rightarrow \{\text{Int} \rightarrow \text{Int}\}
Currying

When Partial Application Matters

\[ f_3 :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ f_3 = \lambda x \rightarrow \text{let } z = \text{expensive} \ x \ \text{in} \ \lambda y \rightarrow y + z \]

• Because of \( \eta \), \( f_3 \) now has arity 2, not 1!
  • \( \text{map \ (f_3 \ 100)} \ [1..10^6] \) recomputes ‘expensive 100’ a million times 😞

\[ f_3' :: \text{Int} \rightarrow \{ \text{Int} \rightarrow \text{Int} \} \]
\[ f_3' = \lambda x \rightarrow \text{let } z = \text{expensive} \ x \ \text{in} \ \text{Clos} (\lambda y \rightarrow y + z) \]

• \( f_3' \) is an arity 1 function; returns a closure \( \{\text{Int} \rightarrow \text{Int}\} \) of an arity 1 function
  • \( \text{map \ (App \ (f_3' \ 100)) \ [1..10^6]} \) computes ‘expensive 100’ only once 😊

\[ \text{Clos} :: (\text{Int} \rightarrow \text{Int}) \rightarrow \{\text{Int} \rightarrow \text{Int}\} \quad \text{App} :: \{\text{Int} \rightarrow \text{Int}\} \rightarrow \text{Int} \rightarrow \text{Int} \]
Functions are *Called*

Not *Evaluated*
Functions are **Called**

\[
x = \text{let } f :: \text{Int } \rightarrow \text{Int} = \text{expensive } 100 \text{ in } \ldots f \ldots f\
\]
Functions are *Called*

\[
x = \text{let } f :: \text{Int} \rightarrow \text{Int} = \text{expensive 100} \text{ in } \ldots f \ldots f\ldots
\]

- When is \text{expensive 100} evaluated?
Functions are **Called**

\[ x = \text{let } f :: \text{Int } \to \text{Int} = \text{expensive 100 in } \ldots f \ldots f \ldots \]

- **When is expensive 100 evaluated?**
  - Call-by-value: first, before binding \( f \)
Functions are \textit{Called} \hfill \textbf{Not \textit{Evaluated}}

\[ x = \text{let } f :: \text{Int } \rightarrow \text{Int} = \text{expensive 100} \text{ in } \ldots f \ldots f \ldots \]

\begin{itemize}
  \item \textbf{When is expensive 100 evaluated?}
  \begin{itemize}
    \item Call-by-value: first, before binding \( f \)
    \item Call-by-need: later, but only once, when \( f \) is first demanded
  \end{itemize}
\end{itemize}
Functions are *Called*

x = let f :: Int ~> Int = expensive 100 in ...f...f...

- When is `expensive 100` evaluated?
  - Call-by-value: first, before binding `f`
  - Call-by-need: later, but only once, when `f` is first demanded
  - Call-by-name: later, re-evaluated every time `f` is demanded
Functions are *Called*

x = let f :: Int ~> Int = expensive 100 in ...f...f...

• When is \texttt{expensive 100} evaluated?
  • Call-by-value: first, before binding \texttt{f}
  • Call-by-need: later, but only once, when \texttt{f} is first demanded
  • Call-by-name: later, re-evaluated every time \texttt{f} is demanded

x' = let f :: Int ~> Int = \y -> expensive 100 \(y\) in ...f...f...
Functions are \textit{Called} \textit{Not Evaluated}

\[ x = \text{let} \ f :: \text{Int} \rightarrow \text{Int} = \text{expensive 100} \ \text{in} \ \ldots f \ldots f \ldots \]

• When is \text{expensive 100} evaluated?
  • Call-by-value: first, before binding \( f \)
  • Call-by-need: later, but only once, when \( f \) is first demanded
  • Call-by-name: later, re-evaluated every time \( f \) is demanded

\[ x' = \text{let} \ f :: \text{Int} \rightarrow \text{Int} = \lambda y \rightarrow \text{expensive 100} \ y \ \text{in} \ \ldots f \ldots f \ldots \]

• \( x = x' \) by \( \eta \), and \( x' \) always follows call-by-name order!
Functions are **Called**

• When is `expensive 100` evaluated?
  • Call-by-value: first, before binding \( f \)
  • Call-by-need: later, but only once, when \( f \) is first demanded
  • Call-by-name: later, re-evaluated every time \( f \) is demanded

\[
x = \text{let } f :: \text{Int} \Rightarrow \text{Int} = \text{expensive 100} \text{ in } ...f...f...
\]

\[
x' = \text{let } f :: \text{Int} \Rightarrow \text{Int} = \backslash y \Rightarrow \text{expensive 100} \ y \text{ in } ...f...f...
\]

• \( x = x' \) by \( \eta \), and \( x' \) always follows call-by-name order!

• Primitive functions are never just *evaluated*; they are always *called*
The Problem With Polymorphism

And Static Compilation
The Problem With Polymorphism

poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)
The Problem With Polymorphism

poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)

• What are the arities of f and g? Counting arrows...
The Problem With Polymorphism

poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)

• What are the arities of f and g? Counting arrows...
  • f :: Int ~> Int ~> a has arity 2
The Problem With Polymorphism

poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)

• What are the arities of f and g? Counting arrows...
  • f :: Int ~> Int ~> a has arity 2
  • g :: Int ~> a has arity 1
The Problem With Polymorphism

poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)

- What are the arities of f and g? Counting arrows...
  - f :: Int ~> Int ~> a has arity 2
  - g :: Int ~> a has arity 1

- But what if a = Bool ~> Bool?
poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)

• What are the arities of f and g? Counting arrows...
  • f :: Int ~> Int ~> a has arity 2
  • g :: Int ~> a has arity 1

• But what if a = Bool ~> Bool?
  • f :: Int ~> Int ~> Bool ~> Bool has arity 3...
The Problem With Polymorphism

poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)

• What are the arities of f and g? Counting arrows...
  • f :: Int ~> Int ~> a has arity 2
  • g :: Int ~> a has arity 1

• But what if a = Bool ~> Bool?
  • f :: Int ~> Int ~> Bool ~> Bool has arity 3...
  • g :: Int ~> Bool ~> Bool has arity 2... oops...
The Problem With Polymorphism

poly :: forall a. (Int ~> Int ~> a) ~> (a, a)
poly f = let g :: Int ~> a = f 3 in (g 5, g 4)

• What are the arities of f and g? Counting arrows...
  • \( f :: \text{Int} \to \text{Int} \to a \) has arity 2
  • \( g :: \text{Int} \to a \) has arity 1

• But what if \( a = \text{Bool} \to \text{Bool} \)?
  • \( f :: \text{Int} \to \text{Int} \to \text{Bool} \to \text{Bool} \) has arity 3...
  • \( g :: \text{Int} \to \text{Bool} \to \text{Bool} \) has arity 2... oops...

• How to statically compile? Is ‘g 5’ a call? A partial application?
Nonuniform Representation

And Static Compilation
Nonuniform Representation

• Primitive types:
Nonuniform Representation

• Primitive types:
  • Int#, Float#, Char#, Array#...
Nonuniform Representation

• Primitive types:
  • Int#, Float#, Char#, Array#...

• Unboxed

And Static Compilation
Nonuniform Representation

• Primitive types:
  • Int#, Float#, Char#, Array#...

• Unboxed
  • Efficient passing

And Static Compilation
Nonuniform Representation

- Primitive types:
- Int#, Float#, Char#, Array#...

And Static Compilation

- Unboxed
- Efficient passing

- Different sizes
Nonuniform Representation

- Primitive types:
  - Int#, Float#, Char#, Array#...

- Unboxed
- Efficient passing

And Static Compilation

- Different sizes
- Different locations
Nonuniform Representation

- Primitive types:
- Int#, Float#, Char#, Array#...

Unboxed
- Efficient passing

And Static Compilation

- Different sizes
- Different locations
- Different levy
Nonuniform Representation

- Primitive types:
  - Int#, Float#, Char#, Array#...
- Unboxed
- Efficient passing
- Different sizes
- Different locations
- Different levity

And Static Compilation

revapp :: forall a b. a -> (a -> b) -> b
revapp x f = f x
Nonuniform Representation

- Primitive types:
- Int#, Float#, Char#, Array#...

```
revapp :: forall a b. a -> (a -> b) -> b
revapp x f = f x
```

```
(++) :: [a]    -> [a]    -> [a]
plusFloat# :: Float# -> Float# -> Float#
```
Nonuniform Representation

- Primitive types:
  - Int#, Float#, Char#, Array#...
- Unboxed
  - Efficient passing
- Different sizes
- Different locations
- Different levity

And Static Compilation

revapp :: forall a b. a -> (a -> b) -> b
revapp x f = f x

(++) :: [a] -> [a] -> [a]
plusFloat# :: Float# -> Float# -> Float#

revapp [0..3] (++) [4..9]) vs revapp 2.5 (plusFloat# 1.5)
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is uniform
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is \textit{uniform}
  • ‘a’ is represented as a pointer
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is uniform
  • ‘a’ is represented as a pointer
  • ‘a’ has arity 0
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is uniform
  - ‘a’ is represented as a pointer
  - ‘a’ has arity 0
- Restriction on quantifiers for all $a::k$. ...
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
  - ‘\(a\)’ is represented as a pointer
  - ‘\(a\)’ has arity 0

- Restriction on quantifiers \(\forall a :: k. \ldots\)
  - Special kinds for unboxed (\#) and non-zero arity (~) types
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is uniform
  • ‘a’ is represented as a pointer
  • ‘a’ has arity 0

• Restriction on quantifiers \( \text{forall } a::k. \ldots \)
  • Special kinds for unboxed (#) and non-zero arity (~) types
    • \( k \) may be ★ or ★→★ but never # or ~
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is \textit{uniform}
  • ‘a’ is represented as a pointer
  • ‘a’ has arity 0

• Restriction on quantifiers \texttt{forall} a::k. ...
  • Special kinds for unboxed (#) and non-zero arity (~) types
  • k may be ★ or ★→★ but never # or ~

• Draconian restriction is unsatisfactory
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

- All polymorphism is *uniform*
  - ‘a’ is represented as a pointer
  - ‘a’ has arity o

- Restriction on quantifiers forall a::k. ...
  - Special kinds for unboxed (#) and non-zero arity (~) types
  - k may be ★ or ★->★ but never # or ~

- Draconian restriction is unsatisfactory
  - Too restrictive: Identical definitions/code repeated for different types (like error :: String -> a)
A Stop-Gap Solution

Uniform Polymorphism in a Nonuniform Language

• All polymorphism is \textit{uniform}
  • ‘a’ is represented as a pointer
  • ‘a’ has arity 0

• Restriction on quantifiers \textit{forall} \ a::k. ...
  • Special kinds for unboxed (#) and non-zero arity (~) types
  • k may be ★ or ★->★ but never # or ~

• Draconian restriction is unsatisfactory
  • \textbf{Too restrictive}: Identical definitions/code repeated for different types (like error :: String -> a)
  • \textbf{Incompatible with kind polymorphism}: \textit{forall} \ k::Kind. \textit{forall} \ a::k. ???
Representation Polymorphism

Kinds As Representations
Representation Polymorphism

• Generalize $a: \star$ to $a: \text{TYPE} \ r$
Representation Polymorphism

• Generalize $a::\star$ to $a::\text{TYPE}$ $r$
  
  • $r::\text{Rep}$ is the *representation* of $a$
Representation Polymorphism

- Generalize $a::\star$ to $a::\text{TYPE} \ r$
  - $r::\text{Rep}$ is the representation of $a$
  - $\star = \text{TYPE} \ \text{Ptr}$
Representation Polymorphism

• Generalize $a::\star$ to $a::\text{TYPE r}$
  • $r::\text{Rep}$ is the representation of $a$
  • $\star = \text{TYPE Ptr}$

```plaintext
revapp x f = f x
```
Representation Polymorphism

• Generalize $a:\Type \to r$
  
  • $r::\text{Rep}$ is the *representation* of $a$
  
  • $\Type \equiv \text{Ptr}$

```haskell
revapp x f = f x

revapp :: forall (r1,r2::Rep) (a::Type r1) (b::Type r2). a -> (a -> b) -> b
```
Representation Polymorphism

• Generalize $a::\star$ to $a::\text{TYPE } r$
  • $r::\text{Rep}$ is the representation of $a$
  • $\star = \text{TYPE } \text{Ptr}$

\[
\text{revapp } x \ f = f \ x
\]

\[
\text{revapp } :: \forall (r1, r2::\text{Rep}) (a::\text{TYPE } r1) (b::\text{Type } r2). \\
\ \ \ \ a \to (a \to b) \to b
\]
Representation Polymorphism

- Generalize \( a : \star \) to \( a : \text{TYPE} \ r \)
  - \( r : \text{Rep} \) is the representation of \( a \)
  - \( \star = \text{TYPE} \ 	ext{Ptr} \)

\[
\text{revapp } x \ f = f \ x
\]

\[
\text{revapp} : \forall (r1,r2::\text{Rep}) (a::\text{TYPE} \ r1) (b::\text{Type} \ r2).
\quad a \rightarrow (a \rightarrow b) \rightarrow b
\]
Representation Polymorphism

• Generalize $a::★$ to $a::\text{TYPE} \ r$
  • $r::\text{Rep}$ is the representation of $a$
  • $★ = \text{TYPE} \ \text{Ptr}$

```
revapp \ x \ f \ = \ f \ x
```

```
revapp :: forall (r1,r2::\text{Rep}) (a::\text{TYPE} \ r1) (b::\text{Type} \ r2).
          a -> (a -> b) -> b
```

```
revapp :: forall (r::\text{Rep}) (a::\text{TYPE} \ \text{Ptr}) (b::\text{TYPE} \ r).
          a -> (a -> b) -> b
```
Representation Polymorphism

- Generalize \( a::\star \) to \( a::\text{TYPE}\ r \)
  - \( r::\text{Rep} \) is the representation of \( a \)
  - \( \star = \text{TYPE} \ \text{Ptr} \)

\[
\text{revapp} \ x \ f = f \ x
\]

\[
\text{revapp} :: \forall (r1,r2::\text{Rep}) (a::\text{TYPE}\ r1) (b::\text{Type}\ r2). a \to (a \to b) \to b
\]

\[
\text{revapp} :: \forall (r::\text{Rep}) (a::\text{TYPE}\ \text{Ptr}) (b::\text{TYPE}\ r). a \to (a \to b) \to b
\]
Representation Polymorphism

• Generalize $a::\star$ to $a::\text{TYPE } r$
  • $r::\text{Rep}$ is the *representation* of $a$
  • $\star = \text{TYPE } \text{Ptr}$

\[
\text{revapp} \ x \ f = f \ x
\]

\[
\text{revapp} :: \forall (r1,r2::\text{Rep}) (a::\text{TYPE } r1) (b::\text{Type } r2).
\quad a \to (a \to b) \to b
\]

\[
\text{revapp} :: \forall (r::\text{Rep}) (a::\text{TYPE } \text{Ptr}) (b::\text{TYPE } r).
\quad a \to (a \to b) \to b
\]
Representation Polymorphism

- Generalize $a::\star$ to $a::\text{TYPE} \ r$
  - $r::\text{Rep}$ is the *representation* of $a$
  - $\star = \text{TYPE} \ 	ext{Ptr}$

```haskell
revapp x f = f x
```

```haskell
revapp :: forall (r1,r2::Rep) (a::\text{TYPE} \ r1) (b::\text{Type} \ r2).
          a -> (a -> b) -> b
```

```
revapp :: forall (r::\text{Rep}) (a::\text{TYPE} \ \text{Ptr}) (b::\text{Type} \ r).
        a -> (a -> b) -> b
```

*Kinds As Representations*
Arity Polymorphism

Kinds As Calling Conventions
Arity Polymorphism

- Generalize $a:TYPE \ r$ to $a:TYPE \ r \ c$
Arity Polymorphism

- Generalize `a::TYPE r` to `a::TYPE r c`
  - `r::Rep` is the *runtime representation* of `a`
Arity Polymorphism

• Generalize $a::\text{TYPE} \ r$ to $a::\text{TYPE} \ r \ c$
  • $r::\text{Rep}$ is the runtime representation of $a$
  • $c::\text{Conv}$ is the calling convention of $a$
Arity Polymorphism

• Generalize $a :: TYPE \ r$ to $a :: TYPE \ r \ c$
  • $r :: Rep$ is the runtime representation of $a$
  • $c :: Conv$ is the calling convention of $a$
• $a :: TYPE \ Ptr \ Call[n]$ says values of $a$ are pointers with arity $n$ (simplified)
Arity Polymorphism

• Generalize $a::\text{TYPE r}$ to $a::\text{TYPE r c}$
  • $r::\text{Rep}$ is the runtime representation of $a$
  • $c::\text{Conv}$ is the calling convention of $a$
  • $a::\text{TYPE Ptr Call[n]}$ says values of $a$ are pointers with arity $n$ (simplified)

poly :: forall $a::\text{TYPE Ptr Call[2]}$. $(\text{Int} \rightarrow \text{Int} \rightarrow a) \rightarrow (a,a)$

poly $f = \text{let } g :: \text{Int} \rightarrow a = f 3 \text{ in (g 4, g 5)}$
Arity Polymorphism

- Generalize \( a::\text{TYPE} \ r \) to \( a::\text{TYPE} \ r \ c \)
  - \( r::\text{Rep} \) is the runtime representation of \( a \)
  - \( c::\text{Conv} \) is the calling convention of \( a \)
  - \( a::\text{TYPE} \ \text{Ptr Call}[n] \) says values of \( a \) are pointers with arity \( n \) (simplified)

\[
poly :: \forall a::\text{TYPE} \ \text{Ptr Call}[2]. (\text{Int} \to \text{Int} \to a) \to (a,a)
\]

\[
poly f = \text{let } g :: \text{Int} \to a = f 3 \text{ in } (g 4, g 5)
\]

- \( f :: \text{Int} \to \text{Int} \to a :: \text{TYPE} \ \text{Ptr Call}[4] \) has arity 4 \((2 + 1 + 1)\)
Arity Polymorphism

- Generalize \( a :: \text{TYPE} \ r \) to \( a :: \text{TYPE} \ r \ c \)
  - \( r :: \text{Rep} \) is the *runtime representation* of \( a \)
  - \( c :: \text{Conv} \) is the *calling convention* of \( a \)
  - \( a :: \text{TYPE} \ \text{Ptr Call}[n] \) says values of \( a \) are pointers with arity \( n \) (simplified)

\[
\text{poly} :: \forall a :: \text{TYPE} \ \text{Ptr Call}[2]. (\text{Int} \to \text{Int} \to a) \to (a, a)
\]

\[
\text{poly} \ f = \text{let} \ g :: \text{Int} \to a = f 3 \ \text{in} \ (g 4, g 5)
\]

- \( f :: \text{Int} \to \text{Int} \to a :: \text{TYPE} \ \text{Ptr Call}[4] \) has arity 4 \((2 + 1 + 1)\)
- \( g :: \text{Int} \to a :: \text{TYPE} \ \text{PTR Call}[3] \) has arity 3 \((2 + 1)\)
Arity Polymorphism

- Generalize $a::\text{TYPE } r$ to $a::\text{TYPE } r \ c$
  - $r::\text{Rep}$ is the runtime representation of $a$
  - $c::\text{Conv}$ is the calling convention of $a$
  - $a::\text{TYPE } \text{Ptr Call}[n]$ says values of $a$ are pointers with arity $n$ (simplified)

```haskell
poly :: forall a::\text{TYPE } \text{Ptr Call}[2]. (\text{Int } \rightarrow \text{Int } \rightarrow a) \rightarrow (a,a)
poly f = let g :: \text{Int } \rightarrow a = f 3 in (g 4, g 5)
```

- $f :: \text{Int } \rightarrow \text{Int } \rightarrow a :: \text{TYPE } \text{Ptr Call}[4]$ has arity 4 $(2+1+1)$
- $g :: \text{Int } \rightarrow a :: \text{TYPE } \text{Ptr Call}[3]$ has arity 3 $(2+1)$

```haskell
revapp :: forall (c::\text{Conv}) (r::\text{Rep})
  (a::\text{TYPE } \text{Ptr } c) (b::\text{TYPE } r \text{ Call}[1]).
  a \rightarrow (a \rightarrow b) \rightarrow b

\text{revapp } x f = f x
```
Arity Polymorphism

- Generalize $a::\text{TYPE} \ r$ to $a::\text{TYPE} \ r \ c$
  - $r::\text{Rep}$ is the runtime representation of $a$
  - $c::\text{Conv}$ is the calling convention of $a$
  - $a::\text{TYPE} \ Ptr \ \text{Call}[n]$ says values of $a$ are pointers with arity $n$ (simplified)

poly :: forall $a::\text{TYPE} \ Ptr \ \text{Call}[2]$. $(\text{Int} \rightarrow \text{Int} \rightarrow a) \rightarrow (a,a)$

poly $f$ = let $g :: \text{Int} \rightarrow a = f 3$ in $(g 4, g 5)$

  - $f :: \text{Int} \rightarrow \text{Int} \rightarrow a :: \text{TYPE} \ \text{Ptr} \ \text{Call}[4]$ has arity $4 \ (2+1+1)$
  - $g :: \text{Int} \rightarrow a :: \text{TYPE} \ \text{PTR} \ \text{Call}[3]$ has arity $3 \ (2+1)$

revapp :: forall $(c::\text{Conv}) \ (r::\text{Rep})$
  $(a::\text{TYPE} \ \text{Ptr} \ c) \ (b::\text{TYPE} \ r \ \text{Call}[1])$.
  $a \rightarrow (a \rightarrow b) \rightarrow b$

revapp $x \ f = f \ x$

  - $f :: a \rightarrow b :: \text{TYPE} \ \text{Ptr} \ \text{Call}[2]$ has arity $2$
Arity Polymorphism

- Generalize $a::\text{TYPE} \ r$ to $a::\text{TYPE} \ r \ c$
  - $r::\text{Rep}$ is the *runtime representation* of $a$
  - $c::\text{Conv}$ is the *calling convention* of $a$
  - $a::\text{TYPE} \ \text{Ptr Call}[n]$ says values of $a$ are pointers with arity $n$ (simplified)

```plaintext
poly :: forall a::\text{TYPE} \ \text{Ptr Call}[2]. (\text{Int} \to \text{Int} \to a) \to (a,a)
poly f = let g :: \text{Int} \to a = f 3 in (g 4, g 5)
```

- $f :: \text{Int} \to \text{Int} \to a :: \text{TYPE} \ \text{Ptr Call}[4]$ has arity 4 ($2 + 1 + 1$)
- $g :: \text{Int} \to a :: \text{TYPE} \ \text{PTR Call}[3]$ has arity 3 ($2 + 1$)

```plaintext
revapp :: forall (c::\text{Conv}) (r::\text{Rep})
          (a::\text{TYPE} \ \text{Ptr c}) (b::\text{TYPE} \ \text{r Call}[1]).
          a \to (a \to b) \to b
revapp x f = f x
```

- $f :: a \to b :: \text{TYPE} \ \text{Ptr Call}[2]$ has arity 2
- $x :: a :: \text{TYPE} \ \text{Ptr c}$ is represented as a pointer
Levity Polymorphism

Call vs Eval, Revisited
Levity Polymorphism

- Code that isn’t called is evaluated
Levity Polymorphism

- Code that isn’t called is evaluated
  - Eval $U :: Conv$ — eager (call-by-value) evaluation, Unlifted values
Levity Polymorphism

- Code that isn’t called is evaluated
  - Eval $U : : \text{Conv}$ — eager (call-by-value) evaluation, Unlifted values
  - Eval $L : : \text{Conv}$ — lazy (call-by-need) evaluation, Lifted values

Call vs Eval, Revisited
Levity Polymorphism

- Code that isn’t **called** is **evaluated**
  - Eval $U : : \text{Conv}$ — eager (call-by-value) evaluation, Unlifted values
  - Eval $L : : \text{Conv}$ — lazy (call-by-need) evaluation, Lifted values
  - Eval $g : : \text{Conv}$ — polymorphic evaluation, with Levity variable $g$
Levity Polymorphism

• Code that isn’t called is evaluated
  • Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  • Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
  • Eval g :: Conv — polymorphic evaluation, with Levity variable g

Int g :: Type Ptr (Eval g) -- boxed, levity-g integers
Levity Polymorphism

- Code that isn’t called is evaluated
  - Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  - Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
  - Eval g :: Conv — polymorphic evaluation, with Levity variable g

\[
\text{Int } g :: \text{Type } \text{Ptr } (\text{Eval } g) \quad \text{-- boxed, levity-g integers}
\]

\[
\text{sum } :: \text{forall } (g1, g2 :: \text{Levity}). [\text{Int } g1] \Rightarrow \text{Int } g2
\]

\[
\text{sum } [] = 0
\]

\[
\text{sum } (x : xs) = x + \text{sum } xs
\]
Levity Polymorphism

• Code that isn’t called is evaluated
  • Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  • Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
  • Eval g :: Conv — polymorphic evaluation, with Levity variable g

Int g :: Type Ptr (Eval g) -- boxed, levity-g integers

sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
sum [] = 0
sum (x : xs) = x + sum xs
Levity Polymorphism

- Code that isn’t called is evaluated
  - Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  - Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
  - Eval g :: Conv — polymorphic evaluation, with Levity variable g

```haskell
Int g :: Type Ptr (Eval g) -- boxed, levity-g integers
sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
sum [] = 0
sum (x : xs) = x + sum xs
```
Levity Polymorphism

• Code that isn’t called is evaluated
  • Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  • Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
  • Eval g :: Conv — polymorphic evaluation, with Levity variable g

Int g :: Type Ptr (Eval g) -- boxed, levity-g integers

sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
sum [] = 0
sum (x : xs) = x + sum xs
Levity Polymorphism

- Code that isn’t called is evaluated
  - Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  - Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
  - Eval g :: Conv — polymorphic evaluation, with Levity variable g

Int g :: Type Ptr (Eval g) -- boxed, levity-g integers

sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
sum [] = 0
sum (x : xs) = x + sum xs
Levity Polymorphism

• Code that isn’t called is evaluated
  • Eval U :: Conv — eager (call-by-value) evaluation, Unlifted values
  • Eval L :: Conv — lazy (call-by-need) evaluation, Lifted values
  • Eval g :: Conv — polymorphic evaluation, with Levity variable g

Int g :: Type Ptr (Eval g) -- boxed, levity-g integers

sum :: forall (g1, g2 :: Levity). [Int g1] ~> Int g2
sum [] = 0
sum (x : xs) = x + sum xs
Static Compilation

To the Machine
Static Compilation

- Only basic types (pointer, integer, float); no polymorphism
Static Compilation

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls
Static Compilation

• Only basic types (pointer, integer, float); no polymorphism
• Only fully saturated functions and calls

poly :: forall a :: TYPE Ptr Call[2].
  (Int# ~> Int# ~> a) ~> (a, a)
poly f = let g :: Int# ~> a = f 3
        in (g 4, g 5)
Static Compilation

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

\[
\text{poly} :: \forall a :: \text{TYPE Ptr Call}[2]. \\
(\text{Int}# \rightarrow \text{Int}# \rightarrow a) \rightarrow (a, a) \\
\text{poly } f = \text{let } g :: \text{Int}# \rightarrow a = f \ 3 \\
in (g \ 4, g \ 5)
\]
Static Compilation

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

```
poly :: forall a :: TYPE Ptr Call[2].
        (Int# ~> Int# ~> a) ~> (a, a)
poly f = let g :: Int# ~> a = f 3
         in (g 4, g 5)
```

poly = \(f::\text{Ptr}) \rightarrow
Static Compilation

- Only basic types (pointer, integer, float); no polymorphism
- Only fully saturated functions and calls

```haskell
poly :: forall a :: TYPE Ptr Call[2].
    (Int# ~> Int# ~> a) ~> (a, a)
poly f = let g :: Int# ~> a = f 3
         in (g 4, g 5)
```

```
poly = \(f::{Ptr}) ->
    let g::{Ptr} = \(x::{I32}, y::{?}, z::{?}) -> f(3, x, y, z)
```
Static Compilation

With Polymorphic $\eta$-Expansion
poly :: forall a::TYPE Ptr Call[Ptr, F64].
   (Int# ~> Int# ~> a) ~> (a, a)
poly f = let g :: Int# ~> a = f 3
        in (g 4, g 5)
Static Compilation

With Polymorphic $\eta$-Expansion

\[
poly :: \forall a::\text{TYPE} ~ \text{Ptr} ~ \text{Call}[\text{Ptr, F64}].
(\text{Int# } \rightarrow \text{Int# } \rightarrow a) \rightarrow (a, a)
\]
\[
poly f = \text{let } g :: \text{Int# } \rightarrow a = f 3
\text{ in } (g 4, g 5)
\]

\[
poly = \backslash(f::\text{Ptr}) \rightarrow
\]
Static Compilation

poly :: forall a::TYPE Ptr Call[Ptr, F64].
        (Int# ~> Int# ~> a) ~> (a, a)

poly f = let g :: Int# ~> a = f 3
         in (g 4, g 5)

poly = \(f::Ptr) ->
        let g::Ptr = \(x::I32, y::Ptr, z::F64) -> f(3,x,y,z)
Static Compilation

poly :: forall a::TYPE Ptr Call[Ptr, F64].
       (Int# ~> Int# ~> a) ~> (a, a)

poly f = let g :: Int# ~> a = f 3
         in (g 4, g 5)

poly = \(f::Ptr) ->
          let g::Ptr = \(x::I32, y::Ptr, z::F64) -> f(3,x,y,z)
               in (\(y::Ptr, z::F64) -> g(4, y, z),
                   \(y::Ptr, z::F64) -> g(5, y, z))

With Polymorphic η-Expansion
Even More

- **Levity Polymorphism**
  - For when evaluation strategy doesn’t matter

- **Compiling Source $\rightarrow$ Intermediate $\rightarrow$ Target**
  - Via kind-directed $\eta$-expansion and register assignment

- **Type system for ensuring static compilation**
  - Of definitions with arity, levity, and representation polymorphism
Kinds capture the details of efficient calling conventions in low-level machine code.